Things to incorporate into the documentation:

Example in conversation 1

* Completely unrelated messages are correctly answered “I could not find any relevant information in the data specification for your message. “
* Unrelated message but there are words that could be mapped: LLM still maps it. The work flow continues as if the user has asked about the data specification.
  + This is of course unintended. During development, I was too focused on fine tuning for the case when user asks about something relevant.
  + I forgot about fine-tuning for the negative case.
* Ambiguous or complex queries that could map to multiple concepts. How does the application handle this? Does it present options to the user?
  + Short answer: I ask the LLM for exactly one item from the data specification. The same word or phrase will not map to multiple items.
  + Longer anser: depends on the LLM used. Smaller LLMs tend to not follow the given rules strictly so even though I ask for exactly 1 item for each word or phrase, it will return multiple items. In this case, I will show the user all of them.
  + Future work: map concepts to more items. Show the user all of them and ask the user to choose.

Explain the case when there are seemingly multiple of the same concepts. Multiple vehicles, multiple fyzicka osoba, …

Point out that each item is added to the substructure only once. Therefore I don’t support more complex queries.

- Do dokumentace napsat omezení, že přidávám každý item (třídu nebo propertu) jen jednou. Takže některé typy dotazů nejsem schopen vyřešit.

- Více tříd má stejný label, ale různé IRI.

Tady asi sloučit v části suggestions. Ale v části substructure to nechat.

- Vyfiltrovat seznam suggested properties, abych nabízel každý unique property jen jednou.

Pokud mám domain i range nějaké property, tak v tom seznamu je dvakrát, jednou že expanduje domain a podruhé že expanduje range.

- Rozmyslet si, jestli nemůžu podpořit přidání třídy vícekrát. Protože třída může mít property vedoucí na sebe (orgán veřejné moci -> má nadřízený orgán -> orgán veřejné moci)

V tuto chvíli moje sémantika znamená, že chci orgán veřejné moci, který má sebe jako nadřízený orgán veřejné moci.

Myslím, že bych to mohl takto upravit.

To, že některý property se ve feature objeví 2x (jednou pod třídou A, kde A je range a podruhé pod třídou B, kde B je domain), tak nemusí nutně být bug.

- Nechal jsem to tak, protože se mi zdálo docela hezké, že je vidět, že ta properta spojuje ty 2 třídy.

**Expected Project Documentation Structure**

The final documentation should be a self-contained, comprehensive report that serves as both a final deliverable and a reference for future work. It must not simply repeat the proposal and specification but rather report on the *implementation* of those documents.

**1. Abstract**

A brief, high-level summary of the entire project. It should concisely state the problem, the solution, the technologies used, and the key findings.

**2. Introduction**

This section should set the context for the project.

* **Problem Statement:** Reiterate the original problem the project aimed to solve.
* **Project Goals:** Clearly state the project's objectives as outlined in the proposal and specification.
* **Contribution:** Describe the specific contributions of this work. This is where you would explain what was built, what new knowledge was gained, and how it advances the field.

**3. Literature Review and Background**

While the proposal likely had a basic literature review, the final document should have a more in-depth one, focusing on the specific technologies and methods used.

* **LLMs and Prompt Engineering:** Discuss the evolution of LLMs and the principles of prompt engineering that were applied.
* **Knowledge Graphs and SPARQL:** Explain the role of knowledge graphs and SPARQL in the context of the project.
* **Related Work:** Mention other projects or tools that have attempted similar tasks and explain how this project is different or builds upon their work.

**4. System Design and Architecture**

This section is the core of the document and must be highly detailed. It should go beyond the abstract design presented in the specification.

* **Overall Architecture:** Present the final, as-built architecture, detailing all components and their interactions. Use diagrams to illustrate the flow of data.
* **Module Descriptions:** Dedicate a subsection to each major module (e.g., LLM Connector, Conversation Service, SPARQL Translation Service). For each module, describe its function, inputs, outputs, and the technologies used in its implementation.
* **Data Model:** Detail the database schema used to store conversation history, data specification substructures, and any other relevant data. Explain the rationale behind the design.
* **API Documentation:** Provide a complete and accurate documentation of the RESTful API endpoints, including request and response formats.

**5. Implementation and Technical Details**

This is where the student proves they have delivered on the project.

* **Technology Stack:** List and justify the final set of technologies and frameworks used for the backend and frontend.
* **Code Structure:** Describe the project's code structure and how it adheres to the modular design.
* **Deployment:** Explain how the application is deployed and configured.

**6. Evaluation and Results**

This is the most critical section for an opponent. It must demonstrate that the project is a success and has met its goals.

* **Performance Metrics:** Present quantitative results. For example:
  + **Translation Success Rate:** What percentage of natural language queries are successfully translated into correct SPARQL queries?
  + **LLM Comparison:** A head-to-head comparison of the performance of different LLMs on a defined set of test cases.
  + **Latency:** The time it takes for the system to generate a SPARQL query and a suggested response.
* **User Study:** If a user study was conducted, report on its findings. Did users find the system intuitive? Did it successfully help them navigate the data specification?
* **Analysis of Results:** Do not just present data. Interpret it. Explain *why* certain LLMs performed better than others, or why certain types of queries failed.

**7. Discussion and Conclusion**

This section should reflect on the project as a whole.

* **Challenges and Solutions:** Describe the major technical challenges encountered during the implementation phase and how they were overcome.
* **Future Work:** Propose logical next steps for the project. This shows an understanding of the project's limitations and potential for growth.
* **Final Conclusion:** A strong concluding statement summarizing the project's success and its value.

**Important Points to Emphasize**

* **Evidence, Not Just Claims:** The documentation must provide *evidence* for every claim. For example, if the project is independent of a specific LLM, show the code structure and provide an example of how a new LLM could be integrated.
* **Adherence to Specification:** The documentation must explicitly link the final implementation back to the original project specification. For each major use case or user story from the spec, the documentation should describe how it was implemented and whether it was successfully delivered.
* **Critical Self-Assessment:** A high-quality report will not shy away from the project's limitations. It should discuss what did not work as expected and what lessons were learned. This demonstrates academic maturity and a deep understanding of the problem space.
* **Reproducibility:** The documentation should include clear instructions on how to set up and run the project from scratch, including all dependencies and configuration details.

Data specification navigator (documentation)

# Abstract

This project addresses the common problem faced by users who are interested in their organization’s data but do not fully grasp the whole domain ontology and the underlying technical structure. The project’s goal is to act as a conversational intermediary, creating a standalone application that allows users to ask questions in plain language. The final implementation successfully guides the user through the data specification and provides users with executable SPARQL queries, but it still has considerable limitations. The full range of SPARQL syntax is not supported in the current implementation. While the solution is a monolithic application, a key aspect is its modular architecture, ensuring future adaptability and independence from any one specific large language models.

# Introduction

This document presents the realization of the project named “Helping users navigate data specifications”. The original project proposal and detailed specification can be found <here>. I will go over to do: add a road map of the document <here>.

## Motivation

The project’s motivation is best understood through a common scenario: Tanya needs some specific data from her organization’s database. She must visit her organization’s “database person” and formulate her question: “I would like to see our employees who started working here this year”. This “database person” will then query the database and give Tanya a list of employees. The “database person” could also ask: “Do you want all the employees or only employees from a specific department?”. Tanya will answer the question and the two of them can continue in an iterative manner to refine Tanya’s query. My project directly addresses this bottleneck by creating a standalone application that acts as a digital intermediary.

The developed application allows the same kind of iterative refinement of the user’s query as in the given example, eliminating the need for a human expert (the “database person” in our previous example).

# System design and architecture

This section is the core of the document and must be highly detailed. It should go beyond the abstract design presented in the specification.

* **Overall Architecture:** Present the final, as-built architecture, detailing all components and their interactions. Use diagrams to illustrate the flow of data.
* **Module Descriptions:** Dedicate a subsection to each major module (e.g., LLM Connector, Conversation Service, SPARQL Translation Service). For each module, describe its function, inputs, outputs, and the technologies used in its implementation.
* **Data Model:** Detail the database schema used to store conversation history, data specification substructures, and any other relevant data. Explain the rationale behind the design.
* **API Documentation:** Provide a complete and accurate documentation of the RESTful API endpoints, including request and response formats.

The application has a client-server architecture, which comprises of a thin-client frontend and a C# backend.

## Frontend

The frontend was developed using Vite, React and Typescript. This combination was chosen to provide a modern, efficient, and type-safe development environment. React helps provide a component-based user interface, which allows for a modular and reusable codebase. The use of **TypeScript** provided static typing, which reduces runtime errors and improves code maintainability, especially when handling complex data structures returned from the backend API. **Vite** was utilized as the build tool for its **Hot Module Replacement** during development and its optimized production builds.

The user interface (UI) was designed as a thin client to ensure it is lightweight and responsive. It is a single-page application that communicates with the backend via RESTful API calls. The application's user interface is divided into two primary views for user interaction.

### Conversation management view

<Screenshot here>

The conversation management view serves as the main entry point for the application, providing a central hub for users to manage their conversations.

#### Core functionalities

**Display All Conversations:** The view retrieves a list of all existing conversations from the backend and displays them in a grid of interactive cards. Each card shows the conversation's **title**, the name of the **data specification** it's based on, and the **last updated** timestamp. If there are not yet any conversations, the view provides the user with a short instruction on how to create a new conversation.

**Create a New Conversation:** Users can create a new conversation in one of two ways:

1. **Manual Creation:** By clicking the "Create new conversation" button, a dialog appears that prompts the user to manually enter the **Dataspecer package IRI** and a title for the new conversation.
2. **Browser redirect:** Conversation creation can also be initiated by a browser redirect from the Dataspecer tool. It detects specific URL parameters (uuid and packageName) to automatically open a new conversation dialog with the Dataspecer package information pre-filled. This feature was implemented to streamline the user experience when navigating directly from the Dataspecer tool.

**Open a Conversation:** Each conversation card features an "Open" button that navigates the user to the dedicated **Conversation View**.

**Delete a Conversation:** Users can permanently delete a conversation by clicking the "Delete" button on its card, which will prompt a confirmation dialog to prevent accidental deletion. Upon confirming the delete action, the application performs an **optimistic update**, immediately removing the card from the UI and then sending a DELETE request to the backend.

#### Technical details

This view is implemented as a React component and it interacts with the backend using RESTful API calls.

The UI provides clear visual cues to the user when interacting with the backend:

* Loading conversations: when fetching conversations data, skeleton cards are displayed to indicate that the app is loading conversations.
* A spinning circle while waiting for the backend call that creates a new conversation.
* Whenever a backend call fails or in case of other errors, the UI displays an error message colored red.

The functionality for redirecting from Dataspecer directly to the conversation creation dialog is currently not live. It is implemented in a forked Dataspecer repository <add link here>.

### Conversation view

<Screenshot [ConversationPage\_initial.PNG]>

<Screenshot [ConversationPage\_message\_sent\_successfully.PNG]>

This view is the primary interface for users to interact with the chatbot, emulating the familiar conversational flow of mainstream LLM services.

#### Core functionalities

**Conversation history:** The view fetches and renders all messages of the given conversation in chronological order.

**User input and submission:** The user can type their natural language query into an input field at the bottom of the screen. Upon submission, the UI immediately updates with the user’s message and it to the backend via a RESTful API call.

**Data specification navigation:** The UI provides two key features to help users navigate the data specification.

* 1. **Mapped item highlighting:** The backend identifies and highlights words in the user’s message that refer to data specification items. The UI renders these words as clickable links, and a pop-up dialog provides a summary of the corresponding item.
  2. **Query refinement via suggestions:** Each chatbot reply may contain suggested items from the data specification. Clicking on a suggestion opens a dialog with a detailed summary of the item, allowing the user to understand its purpose and relevance. Users can select one or more suggested items from the chatbot’s most recent reply to expand their query.

**Display suggested message:** Once the user has confirmed selected suggestions, a suggested message is generated by the backend, and the view will display it above the input field. This verbalization combines the user’s message with the selected suggestions, allowing them to preview the expanded query before submission.

#### Technical details

The Conversation View is a complex, stateful component that orchestrates communication with the backend and manages the dynamic UI.

#### Backend API interaction

When a user sends a message, the frontend initiates a sequence of three API calls to the backend. It first sends a POST request to log the user’s message, followed by a GET request to retrieve the system’s reply. Finally, it makes a third GET request to retrieve the mapped data specification items to display in the right-side panel.

The view renders three distinct message types: welcome, user, and reply messages. The rendering logic for all messages is encapsulated in a separate `MessagesList` React component. This decoupling ensures the code is extensible and easy to maintain, which proved valuable during development as the message structure evolved.

## Backend

The backend is a monolithic application written in C# using the Minimal API framework. While deployed as a single unit, the system is designed with a layered and highly modular architecture to ensure it is extensible and easy to maintain.

The backend is logically divided into three layers:

1. **Business core layer:** This is the heart of the application, containing all the core business logic. It orchestrates the flow of data, handles conversation state, and processes user requests by interacting with the connectors.
2. **Connectors layer:** This layer is a set of interfaces that abstract communication with external systems. This design ensures that the business core is not directly dependent on any specific external service.
3. **External systems layer:** These are the third-party services that the application depends on, such as various Large Language Models (LLMs) and the Dataspecer tool.

### Connectors

This layer defines two main interfaces: ILlmConnector and IDataspecerConnector. This design allows for different implementations to be swapped in without affecting the core business logic.

#### Dataspecer connector

The sole purpose of this connector is to retrieve the data specification files (DSV and OWL) from Dataspecer. It is responsible for fetching these files and returning their contents to the business core layer.

#### LLM connector

To demonstrate the system’s independence from any specific LLM, I have created two separate implementations: OllamaConnector and GeminiConnector. Each connector is responsible for a specific LLM and its unique requirements, including prompting and response processing.

#### Core logic: LLM interaction

The system’s interaction with the LLMs is handled through a pair of dedicated components for each connector:

* Prompt constructor (ILlmPromptConstructor): This component transforms the input from the business core into a format optimized for its specific LLM. For example, the LlamaPromptConstructor provides a relevant subset of the data specification to the smaller llama3.3:70b, whereas the GeminiPromptConstructor feeds the entire OWL content of the data specification into its prompt template.
* Response Processor (ILlmResponseProcessor): This component receives the raw output from the LLM, which is typically in JSON format as requested in prompt templates, and converts it into the data structures that the business core layer works with. This ensures the core logic can process the LLM’s response consistently, regardless of which LLM was used.

### Business core layer

This layer is the heart of the backend, containing the core business logic. It is composed of three main services: a Data Specification Service, a Conversation Service, and a SPARQL Translation Service. Each service has a single, well-defined responsibility, which ensures a strong separation of concerns.

To make the system easier to maintain and debug, only the Data Specification Service and the Conversation Service have the right to store data to the database, ensuring a single point of data entry for persistence. An exception to this rule is the response processor class, which is allowed to fill in the summary for each data specification item whenever the LLM returns it. All other classes can still access the database for reading but must not write to it.

#### Data specification service

This service is responsible for retrieving and processing the data specification that the user wants to work with. It utilizes the Dataspecer connector to retrieve the DSV or OWL files and then extracts the data specification items from those files for use by the other services.

#### Conversation service

This is the most complex service, managing all aspects of the conversation. Its responsibilities include:

* **Conversation creation:** It creates a new conversation and generates an initial welcome message. This message contains a summary of the data specification and suggests possible starting points for the user.
* **Message processing:** When a user sends a new message, this service adds it to the conversation. It then processes the message by mapping the user’s natural language to the data specification items and generating suggestions for the user.
* **Reply generation:** The service generates the chatbot's reply message, which includes the mapped items, suggestions, and the SPARQL query.
* **Message preview:** It can verbalize a "suggested message" for the user after they have selected suggested items, providing them with a clear, natural-language preview of the expanded query.

The data specification items successfully mapped during the conversation are stored in the conversation itself, forming a mapped substructure. This substructure represents the relevant subset of the full data specification and serves as the primary input for the SPARQL translation service.

#### SPARQL translation service

This service is solely responsible for generating a SPARQL query from the mapped substructure. This translation happens only once, when the chatbot's reply message is being generated, ensuring that the heavy lifting of query generation is done efficiently on the backend before the reply is sent to the frontend.

### Controllers

The final component of the backend are the controllers: IDataSpecificationController and IConversationController. These controllers serve as the entry point for all incoming requests from the frontend and act as an interface to the business core.

Their responsibilities include:

* **Request validation:** They validate incoming data from the frontend, ensuring the data is in the correct format and that any referenced resources, such as conversations or data specifications, exist in the database.
* **Service orchestration:** They orchestrate the business logic by calling the appropriate methods on the service layer. For example, the `ProcessIncomingMessage` method in the IConversationController would coordinate calls to the `AddUserMessageAsync` and `GenerateReplyMessageAsync` methods on the IConversationService.
* **Response adaptation:** They adapt the results from the business services into the specific data format the frontend expects, ensuring a clear and consistent API contract.

## Backend implementation details

This chapter provides a deeper dive into the technical implementation of the backend modules.

### Connectors

To do: Add some text here.

#### Dataspecer connector

The DataspecerConnector is the concrete implementation of the IDataspecerConnector interface. It uses the dotnet HttpClient to download the Dataspecer package specification from a configurable endpoint. It then retrieves either the en/dsv.ttl or en/model.owl.ttl file, depending on the method called. The default endpoint is https://tool.dataspecer.com/api/experimental/output.zip?iri=, but this can be changed by modifying the Env:Dataspecer:Endpoints:DownloadDocumentation environment variable. This configurability is particularly useful for local Dataspecer deployments.

#### LLM connectors

Two distinct LLM connectors were implemented.

OllamaConnector: This connector uses the OllamaSharp library to interact with a locally deployed LLM instance. By default, it sends prompts to llama3.3:70b at localhost:11434. The connector, as well as its associated prompt constructor and response processor (LlamaPromptConstructor and LlamaResponseProcessor), are specifically tuned for this model. While the model can be changed via the Env:Llm:Ollama:Model environment variable, a different model may not adhere to the expected output format, causing the response processor to fail (e.g., Deepseek's <thinking> block is not handled).

GeminiConnector: This connector is implemented using the Google\_GenerativeAI library and requires a valid API key from the ./Secrets/Gemini\_api-key.txt file. For the current build, this connector has been commented out of the dependency injection configuration, as the system is only set up to use one LLM at a time. The code is structured to allow easy swapping between connectors but lacks a mechanism to support multiple LLMs simultaneously, a decision made to prioritize development on core features.

#### LLM prompt templates

The system uses six distinct prompt templates, stored as plain text files, to handle different operations. They are called in the following order:

* welcome\_message\_data\_specification\_summary.txt: Generate a summary about the data specification and suggest some starting points for the user.
* map\_to\_data\_specification.txt: Maps a user's natural language message to the data specification to identify relevant items.
* get\_suggested\_items.txt: Suggests additional data specification items that the user might find useful.
* generate\_suggested\_message.txt: Verbalizes a "suggested message" by incorporating a user's selected suggestions into their original message.
* map\_to\_substructure.txt: Maps the user's confirmed suggested message to the conversation's mapped substructure, which is used for highlighting on the frontend.
* summarize\_data\_specification\_items.txt: Generates a short summary for mapped or suggested items before they are displayed to the user.

All templates are in Markdown format and follow a similar structure: they begin by assigning a role to the LLM, followed by a description of the inputs, the task, and the required output format.

#### Key prompting strategies for llama3.3:70b vs Gemini

The smaller size of llama3.3:70b required a different prompting strategy compared to larger models like Gemini.

* Data specification handling: Instead of providing the entire OWL file, the Llama prompts pass a flattened JSON list of relevant data specification items to reduce token size and improve performance. For some operations (e.g., generating a suggested message), the data specification is omitted entirely. For others, like suggesting items, a "local area" around the mapped substructure is provided to the model.
* Consistent data format: To reduce the cognitive load on the LLM, the input data and the requested output are both in a JSON format.
* Explicit output specification: While Gemini prompts list output fields in a list, Llama prompts use an example JSON object or array to explicitly define the output shape.
* Rule reinforcement: Smaller models are more likely to "forget" strict rules in long prompts. Llama prompts repeat key rules (e.g., returning only a raw JSON object) multiple times to ensure the model adheres to the format.

#### Response processing

Assuming the LLM provides a response in the expected format, the response processor, implemented using dotnet JsonSerializer, parses the JSON into a temporary object. It then validates the data against the database and creates the necessary classes for the business core layer. A common issue is that LLMs often wrap JSON output in backticks (e.g., ```json [...]```), so the response processor is designed to automatically detect and remove these characters.